Use of the Message Passing Interface in a Real-Time SONAR Beamformer

Daniel H. Kim
Naval Undersea Warfare Center Division Newport

Abstract

The Message Passing Interface (MPI) is the standard inter-process communication library developed to implement a portable, efficient, and scalable parallel program. In this paper, I evaluate the MPI by implementing the sonar algorithm called "Range Focused k-ω Beamforming" and conducting several experiments. The general overview of MPI is described with its history and current status. A description of the parallelization approach utilized is provided along with the experimental results on three parallel machines: Cray T3D, SGI Origin 2000, and embedded Mercury RACE system.

I conclude that the MPI is a practical tool to implement a portable, efficient, and scalable parallel program for a real-time system, however, it is not easy to write an efficient program. MPI always includes communication overhead but the impact of this overhead may be reduced by the use of nonblocking communication routines, double buffers, and increased problem size to lower the communication/computation ratio. The visualization tool called "upshot" is helpful to finely tune the program to achieve load balancing.

Introduction

A beamformer is an essential part of a sonar (SOund NAvigation and Ranging) and radar (RAdio Detecting And Ranging) system that combines the sensor inputs in an appropriate fashion to detect the targets of interest in real-time. There have been many beamforming algorithms developed and more are currently being developed to achieve increased accuracy and efficiency. Even one of the most efficient algorithms, however, is computationally very intensive and requires a great amount of memory. It is called "Range Focused k-ω Beamforming" [10]. Naturally, parallel computers are needed to support its computational demand.

Over the past few years, many parallel computers have been developed and deployed as a platform for sonar systems. However, there have been a number of challenges confronted in programming parallel computers. The software developed for one system could not be ported to new and improved hardware without great expense mainly because the inter-process communication schemes and languages are not standard across different architectures. Also the need for scalability of the parallel program has arisen to accommodate different problem sizes, such as a different number of sensors, increased look directions or ranges, etc. Portability and high performance are conflicting attributes. A parallel program must not depend on machine specific operations designed for high performance to be portable, however, it must achieve the high performance without sacrificing portability.

Three different parallel programming paradigms are currently being widely used in writing parallel programs: data parallelism, shared memory, and message passing. High Performance Fortran (HPF) [5][9], a set of extensions to Fortran 90, is the most popular *data-parallel* language, in which the compiler generates a parallel program from a serial program. The data is distributed over the processors and each processor executes the same instructions on its portion of the data. This paradigm is very well suited to Single Instruction Multiple Data (SIMD) machines and for algorithms with very regular data structures. If the algorithm, however, uses a dynamic and sparse data structure, then it is not easy to balance the loads because the data mapping is done at compile time. *Shared-memory* [4][9] is a paradigm for both shared memory and distributed memory systems, in which the processes communicate by directly sharing data as if the data existed in a global memory space. The main advantage usually associated with the shared memory paradigm is that the application programming interface is quite simple and therefore the complexity of developing parallel applications is reduced. Understanding and managing the locality of the data, however, becomes more difficult and writing deterministic parallel programs is also difficult. *Message-passing* [1][2][6][7][8] is a paradigm for both shared memory and distributed memory systems and is based on the explicit sending and receiving of messages.

Of these methods, the MPI standard is currently the most widely used method for programming parallel systems since it is designed to achieve portability, high performance, and scalability [1][2][6][7][8][9]. The MPI program is portable since there are many implementations available either freely or commercially and is efficient because virtually all parallel computer vendors supply their own optimized versions of MPI [11].

The rest of this paper is organized as follows. Section 2 gives a general overview of MPI with its history and current status. In Section 3, the parallelization effort of the “Range Focused k-ω Beamforming” algorithm using MPI is described. In Section 4, the result of a number of performance measurements is presented. In Section 5, the lessons I have learned are described with a future plan.

Message Passing Interface

History

The message-passing paradigm for parallel programming is widely used among parallel computer vendors and users. There have been a number of libraries developed and used to achieve portability between parallel applications [1][2][6]; PICL, PVM, PARMACS, p4, Chameleon, Zipcode, TCGMSG, and Express. These libraries have competed with one another and each vendor has focused on making its product unique. The syntactic differences and numerous minor incompatibilities made it difficult to port applications from one computer to another [1][2][6].

In April 1992, many researchers from vendors and universities began the effort to create a standard to enable portability of message-passing programs at a workshop on Standards for Message Passing in a Distributed-Memory Environment [1][2]. As a result of that workshop, the Message Passing Interface (MPI) Forum was organized at the Supercomputing ’92 Conference. The first version, Version 1.0, of the MPI Standard was completed in May 1994, and it has evolved since.

From the very beginning of the MPI Forum while the MPI Standard definitions were still evolving, Gropp and Lusk started developing the very first implementation of MPI, MPICH [1].

They could implement the proposed definitions very quickly because the large portion of MPICH was borrowed from the existing portable libraries; p4, Chameleon, and Zipcode. This approach quickly exposed the problems that the proposed specification might pose for other implementers and also provided the experimenters with a way to try ideas being proposed for MPI before they became fixed into the standard. This unique approach made a complete, portable, and efficient implementation of MPI available when the MPI Standard was released in May 1994. The MPICH provided an Abstract Device Interface (ADI) to achieve portability and high performance among different parallel architectures. Consequently, individual vendors and users have taken advantage of this interface to implement their own optimized implementations of it in a short period of time.

Overview of MPI

MPI is a standard message-passing interface for parallel programs. MPI is not a new programming language, rather it is a library of definitions and functions that can be used by C or Fortran programs. The primary goal of the MPI specification is to help the users implementing a portable, scalable, and efficient parallel program. MPI supports both the *multiple program multiple data* (MPMD) model in which each process executes a different program on different data sets and the *single program multiple data* (SPMD) model in which each process executes a same program on different data sets [9].

In a typical MPI program, a fixed set of processes is created at program initialization, and one process is created per processor. A subset of processes can be grouped together to form a *communicator* that can send messages between the members. Each communicator is composed of a *group* and a *context*. A group is an ordered collection of processes where each process in the group is assigned a unique rank. A context is a system-defined object that is associated with a group in a communicator. Two distinct communicators have different contexts, even if they have identical underlying groups. MPI provides an additional way, called a "virtual *topology*", for associating different addressing schemes with the processes belonging to a group. There are two types of virtual topologies in MPI, Cartesian or grid and graph topology. Even though there may be no simple relationship between the process structure in a virtual topology and the actual physical structure of the parallel system, virtual topologies help to relate the application semantics to the message passing semantics in a convenient and efficient way [2][6][7].

MPI provides basically two kinds of communication schemes, collective or global and point-to-point communication operations. A group of processes can use collective communication operations to send or receive messages to or from every process in a group and also to perform user defined or predefined global operations such as summation, product, maximum, minimum, etc. Processes can use point-to-point communication operations to send messages from one process to another [2][6][7][8].

The set of point-to-point communication routines is very rich and forms the core of MPI. They are categorized into blocking and nonblocking routines [2][6][7][8]. Blocking send and receive routines do not return until it is safe to alter the message buffer. Nonblocking send and receive routines may return while the message buffer is still volatile, and therefore it is the programmer's responsibility to ensure that the message buffer is not changed until it is guaranteed that this will not corrupt the message. Nonblocking routines always come in two parts: a posting routine to start the

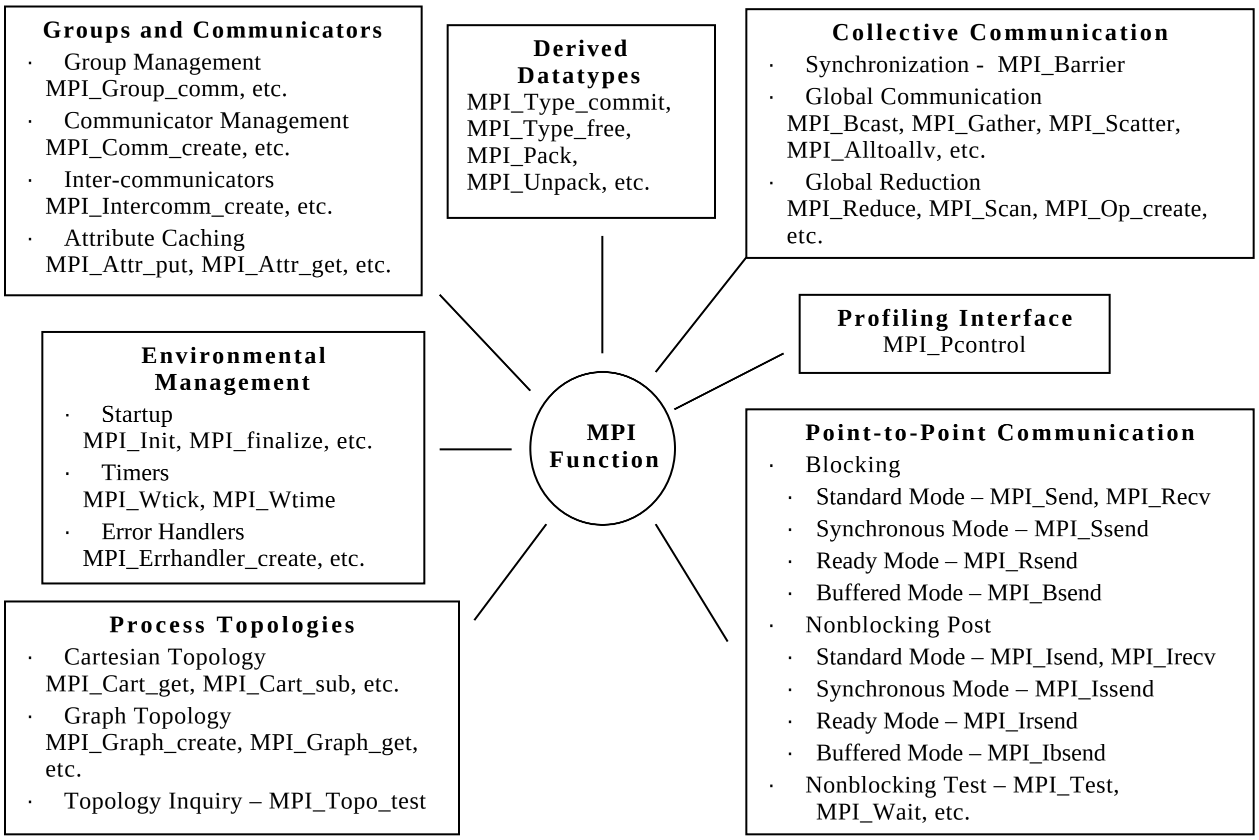

Groups and Communicators
· Group Management MPI_Group_comm, etc.
· Communicator Management MPI_Comm_create, etc.
· Inter-communicators MPI_Intercomm_create, etc.
· Attribute Caching MPI_Attr_put, MPI_Attr_get, etc.
Derived Datatypes
MPI_Type_commit, MPI_Type_free, MPI_Pack, MPI_Unpack, etc.
Collective Communication
· Synchronization - MPI_Barrier
· Global Communication MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Alltoallv, etc.
· Global Reduction MPI_Reduce, MPI_Scan, MPI_Op_create, etc.
Profiling Interface
MPI_Pcontrol
Environmental Management
· Startup MPI_Init, MPI_finalize, etc.
· Timers MPI_Wtick, MPI_Wtime
· Error Handlers MPI_Errhandler_create, etc.
MPI Function
Point-to-Point Communication
· Blocking
· Standard Mode – MPI_Send, MPI_Recv
· Synchronous Mode – MPI_Ssend
· Ready Mode – MPI_Rsend
· Buffered Mode – MPI_Bsend
· Nonblocking Post
· Standard Mode – MPI_Isend, MPI_Irecv
· Synchronous Mode – MPI_Issend
· Ready Mode – MPI_Irsend
· Buffered Mode – MPI_Ibsend
· Nonblocking Test – MPI_Test, MPI_Wait, etc.
Process Topologies
· Cartesian Topology MPI_Cart_get, MPI_Cart_sub, etc.
· Graph Topology MPI_Graph_create, MPI_Graph_get, etc.
· Topology Inquiry – MPI_Topo_test

Figure 1. General Categories of MPI Functions

operation and a testing routine to complete the operation. Nonblocking routines overlap communications and computations, consequently they provide dramatic improvements in the performance of message passing programs.

There are four different point-to-point communication modes for send operations; standard, synchronous, ready, and buffered [2][6][7][8]. In standard mode, a message may be sent regardless of whether a corresponding receive has been initiated and it is up to the system to decide whether messages should be buffered. In synchronous mode, a send is the same as in standard mode, except that it will not complete until a corresponding receive has started, therefore system buffering is not required. In ready mode, a message may be sent only if a corresponding receive has been initiated, thus the ready send is erroneous if the corresponding receive has not been initiated. In buffered mode, a send is the same as in standard mode, except that its completion does not depend on the occurrence of a corresponding receive and the user must explicitly allocate the buffer to ensure that the system resources are not exhausted during communications. There is only a standard mode for receive operations, in which a receive call may start whether or not a corresponding send has been initiated.

In summary, MPI is comprised of 129 functions and definitions [7]. As shown in Figure 1, MPI functions can fall into 7 categories: environmental management, groups and communicators, process topologies, derived datatypes, profiling interface, point-to-point communication, and collective communication functions.

Current Status

Currently MPI is supported by virtually all parallel computer vendors, a network of UNIX or Windows NT workstations, and embedded systems. The MPI Forum has continued evolving the definition to resolve deficiencies in the current standard and add new and powerful functionalities. The MPI-2 standard was finalized in July, 1997 [11]. MPI-2 provides the standard for dynamic programming management, one-sided communications, parallel I/O, extended collective operations, bindings for Fortran 90 and C++, and interlanguage interoperability. Furthermore, other groups formed MPI/RT (Message Passing Interface/Real Time) Forum and have been meeting regularly to design an MPI for real-time programming [11].

Parallel Design Description

This beamforming algorithm [10] like any other algorithm consists of three parts: input, beamforming, and output. Only the beamforming part is parallelized because the input and output stages must be serial. An input processor receives the input data from an external source and distributes them to beamforming processors. An output processor collects the output data from beamforming processors and sends them to an external display device. The beamforming part [10] mainly consists of three steps: transforming input data into frequency and wavenumber domain using a two dimensional Fast Fourier Transform (2DFFT); generating beam output using interpolation; and transforming output back to the time domain using an Inverse FFT (IFFT).

To take advantage of a parallel architecture, two processors are dedicated to an input and output processor, respectively. The remaining processors are utilized as beamforming processors. The parallelization of the beamforming algorithm was approached through domain decomposition [9] in which the data was partitioned. This leads to the principal employment of a *Single Program Multiple Data* (SPMD) model, in which N-2 processors (N is a total number of processors) work on a different portion of the input data set. This approach automatically balances the loads among the N-2 processors. It should be noted, however, that the whole program itself can be viewed as a *Multiple Program Multiple Data* (MPMD) model since all three groups of processors; input, output, and beamforming, execute different instructions concurrently. The principle responsibility of the input processor is to receive input data set from an external source and distribute them to N-2 beamforming processors. The N-2 beamforming processors are tasked with performing two dimensional FFT, beam interpolation, and IFFT to produce the beams. The output processor is responsible for collecting the output data from N-2 beamforming processors and sending them to an external display device.

Each beamforming processor is given an approximately equal number of time series from an input processor. The received data are then transformed to the frequency domain using temporal FFTs. Here a real FFT is used to take advantage of a fact that the input data are real. To perform the spatial FFT, the data must be redistributed among N-2 beamforming processors because each beamforming processor now has the intermediate results required by others. The MPI collective communication routine, **MPI_Alltoallv**, is used to distribute and gather the appropriate data in each

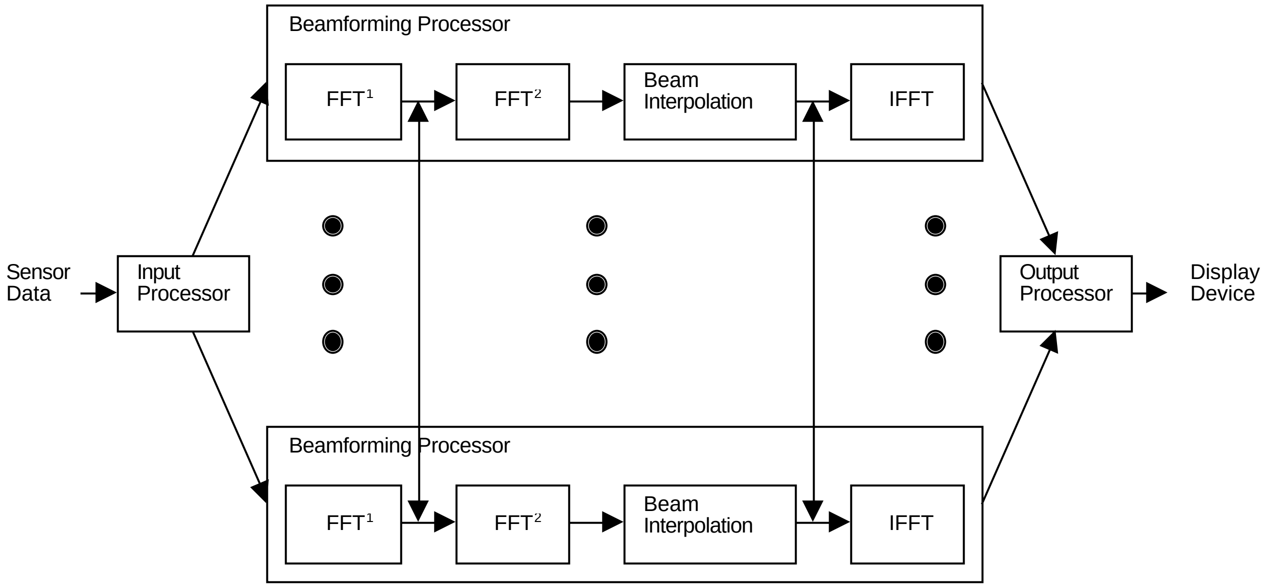


Figure 2. Block diagram of the parallel implementation of “Range Focused k-ω Beamformer”. FFT^1 and FFT^2 represent temporal FFT and spatial FFT, respectively.

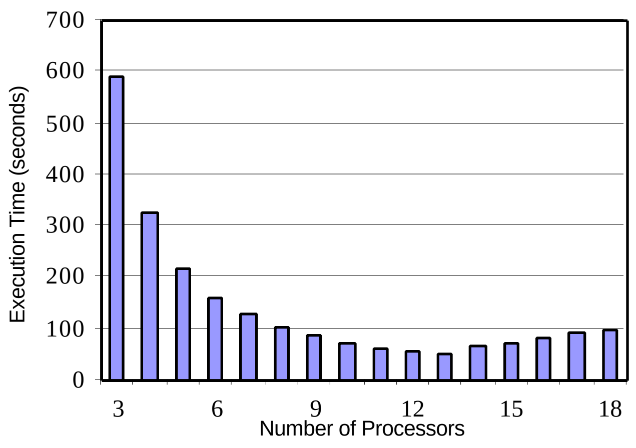
beamforming processor. The gathered data are transformed to wavenumber domain using spatial FFTs. The core of the beamforming algorithm, beam interpolation, is performed to produce the beam output. The output is redistributed and gathered again using another MPI collective communication routine, **MPI_Alltoallv**, for the IFFT operation where the beamformer output is transformed back to the time domain. Figure 2 depicts a block diagram of the parallel implementation of the beamformer.

Using two collective communication routines within the beamforming processors looks very odd and inefficient; however, a single processor could not perform 2DFFT simply because the physical memory was limited. Also one processor could not keep up with the real time requirement. In addition, the communication overhead is hidden by the intensive computation for some of the larger deployed arrays, and it is allowed within the real time requirement for the smaller deployed arrays.

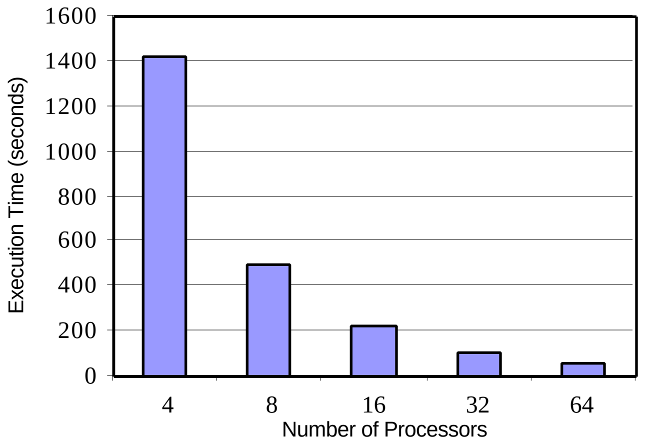
The profiling program called “upshot” [3][6][8] is used to check how the loads are balanced among beamforming processors. The nonblocking MPI communication calls, **MPI_Issend** and **MPI_Irecv**, are used to transfer the data between the input, beamforming, and output processor groups. These nonblocking calls are safely completed with **MPI_Test** and **MPI_Wait** routines. While the beamforming processors are producing their portions of outputs, the input and output processors are concurrently receiving and sending the data sets. The use of double buffering in all three groups reduces the delay introduced by waiting for other processors to finish.

Evaluation

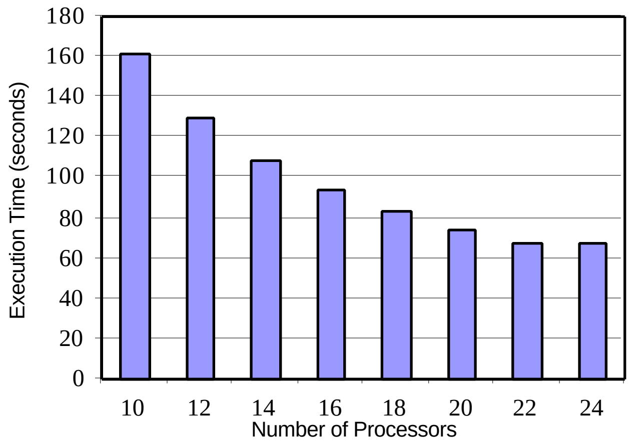
All of the parallelization experiments were performed on three parallel platforms, Cray T3D, SGI Origin 2000, and Mercury RACE system. The Cray T3D, a distributed-memory parallel



(a) SGI Origin 2000



(b) Cray T3D



(c) Mercury RACE

Figure 3. Results of Scaling Experiments

supercomputer, is equipped with 64 DEC Alpha processors with a clock speed of 150 MHz each. The SGI Origin 2000, a shared-memory parallel supercomputer, is equipped with 18 MIPS R10000 processors with a clock speed of 250 MHz each. The Mercury RACE system, a VME based embedded processor, is equipped with 24 Motorola PowerPC processors with a clock speed of 200 MHz each interconnected through a 160 Mbytes/second switched network called RACEway. Each system has a different implementation of MPI; a freely available MPI called MPICH [1] on the Cray T3D, a freely available MPI called RACEMPI on the Mercury RACE system, and SGIs optimized version of MPI on the SGI Origin 2000. All parallel programs were written in ANSI C and MPI.

Initially I implemented the beamforming program on the SGI Origin 2000. The *portability* of the MPI program was verified by moving the program to different platforms, the Cray T3D and the Mercury RACE system, recompiling the program, and running it without any problem.

The *scalability* is defined to be the amount of speedup achieved as additional processors are added when executing the application [8][9]. Ideal scaling, in this situation, would see the parallel application execute N times faster on N processors than the serial application on a single processor. In practical terms, the speedup is limited by the communication costs and Amdahl's Law. Amdahl's Law [9] states: if the sequential component of an algorithm accounts for 1/*s* of the program's execution time, then the maximum possible speedup that can be achieved on a parallel computer is *s*. The scalability of the MPI program was examined using a different number of processors while keeping the problem size unchanged to analyze how much speedup can be achieved. The problem parameters used in this experiment were 8 focus ranges, 400 sensors, 401 beams, 629 frequency bins, a temporal FFT size of 2048 points, and 100 seconds of input data. Note that two processors were always dedicated to input and output processors no matter how many processors were used. The SGI Origin 2000 was most flexible so I could use any number of processors from 3 to all 18 in the experiments. The Cray T3D has a restriction in which only a power of 2 of processors can be used. Unlike SGI and Cray which are equipped with a large amount of memory, Mercury RACE system contains only 16 Mbytes on each processor. This limited the minimum number of processors to 10. As shown in Figure 3, almost linear speedup was achieved on all three platforms to a certain point. On the SGI Origin 2000, the degraded performance after 13 processors was noticeable because the communication overhead becomes more expensive than the actual computation as shown in Figure 3(a). The Cray T3D system handled better in terms of the scalability as shown in Figure 3(b). It did not show any degraded performance with up to 64 processors. The Mercury RACE system started showing the communication cost when 24 processors were used, the execution time did not decrease even though the number of processors was increased as shown in Figure 3(c). From this experiment, the beamformer needs at least 8 processors on the SGI Origin 2000, 32 processors on the Cray T3D, and 15 processors on the Mercury RACE to satisfy the real-time constraint.

The *efficiency* of MPI program was measured by calculating the FLoating point OPerations per Second (FLOPS) required in “Range Focused k-ω Beamforming” algorithm and comparing it with the empirically measured FLOPS of each machine. The empirically derived machine speeds were 42.0 MFLOPS on each Cray processor, 165.5 MFLOPS on each SGI processor, and 59.5 MFLOPS on each Mercury processor. These speeds were measured by using a simple program that measures the execution time of one million floating point multiply/add operations. The beamforming algorithm basically consists of three steps, a 2 dimensional Fast Fourier Transform (2DFFT), beam interpolation, and an Inverse Fast Fourier Transform (IFFT).

For the same problem size used in scalability experiment, the 2DFFT step requires approximately 33.4 Millions of FLoating point OPerations (MFLOP), the interpolation step requires 308.4 MFLOPs, and the IFFT step requires 70.5 MFLOPs. By summing these, the total number of floating point operations required in the beamforming algorithm is about 412.3 MFLOPs. This number represents the necessary work to process about 0.9 seconds of input data. From the experiment as shown in Figure 3, to process 100 seconds of input data, the beamformer took 104.0 seconds on SGI Origin 2000 using 8 processors, 220.2 seconds on Cray T3D using 16 processors, and 94.0 seconds on Mercury RACE system using 16 processors. Therefore the achieved Millions of FLoating point OPerations per Seconds (MFLOPS) per processor can be calculated on three platforms as follows. For the SGI Origin 2000, $\frac{412.3 \text{ MFLOP}/0.9 \text{ seconds}}{104.0 \text{ seconds}/100 \text{ seconds}} = 440.5 \text{ MFLOPS}$ per 8 processors. Therefore it takes about 73.4 MFLOPS per processor (440.5 MFLOPS / 6 processors) where the measured speed of a chip is 165.5 MFLOPS, achieving about 44.4 % of measured performance. For the Cray T3D, $\frac{412.3 \text{ MFLOP}/0.9 \text{ seconds}}{220.2 \text{ seconds}/100 \text{ seconds}} = 208.0 \text{ MFLOPS}$ per 16 processors. Therefore it takes about 14.9 MFLOPS per processor (208.0 MFLOPS / 14 processors) where the measured speed of a chip is 42.0 MFLOPS, achieving about 35.5 % of its measured performance. For the Mercury RACE system, $\frac{412.3 \text{ MFLOP}/0.9 \text{ seconds}}{94.0 \text{ seconds}/100 \text{ seconds}} = 487.4 \text{ MFLOPS}$ per 16 processors. Therefore it takes about 34.8 MFLOPS per processor (487.4 MFLOPS / 14 processors) where the measured speed of a chip is 59.5 MFLOPS, achieving about 58.5 % of measured performance. From the efficiency experiment, the MPI parallel program achieves about 35~60 % of the measured performance of each processor. Note that I am not considering processors dedicated to input and output functions or processor intercommunication when I estimate the number of floating operations required in the beamforming program.

One thing has to be noted here. I had full control of the number of users on the Mercury RACE system, but not on the SGI and Cray systems. Even though I tried to conduct the experiments while the number of users was at a minimum, the execution time on Cray and SGI might not reflect the actual execution time of the MPI program because it might be affected by other users' activities.

Summary / Conclusion

The MPI is a standard inter-process communication library developed to implement a portable, scalable, and efficient parallel program. To evaluate MPI, one of the most efficient but computationally demanding sonar algorithm called "Range Focused k-ω Beamforming" has been implemented using MPI. I demonstrated that the algorithm could be parallelized and implemented on any parallel platform that supports MPI. Here is what I learned:

· Using domain decomposition, which employs a SPMD design, one can effectively parallelize the beamforming algorithm.
· The MPI parallel program is truly portable. The only thing I had to do was specify the path to the MPI library and recompile the program. Even though I tested the beamformer program on only three parallel platforms, there is no doubt that it will run on any parallel platform that supports MPI.
· The MPI parallel program is scalable to various numbers of processors with an appropriate speedup if the computation burden is large enough to compensate for the communication time. However, as the number of processors increased, the communication overhead greatly limits the speedup experienced and even degrades it.
· The MPI parallel program is efficient enough to be feasible on a real-time system. The beamformer program achieved about 35~60 % of the measured performance on each processor.
· It is not easy to write an efficient parallel program using MPI because the programmer has to consider many small details related to intercommunication. The MPI routines for nonblocking communication reduce the communication time, but they are very dangerous and have to be used very cautiously in order not to destroy the useful data. The use of double buffering increased the throughput.
· The visualization tool called "upshot" program is very helpful to check the load balancing and any inefficiency. Using this tool the program can be finely tuned to achieve the optimum throughput.

In summary, the MPI is a practical tool to implement a portable, efficient, and scalable parallel program. The MPI based parallel program proved to be portable to other parallel platforms, efficient enough to be feasible on a real-time system, and scalable to various number of processors with an appropriate speedup achievement. It should be noted that MPI has communication overhead and that increasing the problem size to lower the communication/computation ratio may reduce the impact of this overhead.

In the future, I plan to port the beamformer to other parallel platforms such as a network of personal computers running Windows NT or Linux connected through Myrinet. Also, I am currently implementing another beamforming algorithm called, "Adaptive Range Focused k-ω Beamformer" using MPI.

Acknowledgments

This research was done under the project CHSSI SIP-2 (Common High-Performance Software Support Initiative Sonar Signal and Image Processing) sponsored by Department of Defense High Performance Computing Modernization Office (DoD HPCMO). I would like to acknowledge our project members, Robert Bernecky and Marvin Kuznitz, for their helpful suggestions and comments. Especially R. Bernecky's "Range Focused k-ω Beamformer on a Line Array" paper [10] and his implementation of the input function.

References

[1] Gropp, W., Lusk, E., Doss, N., and Skjellum, A., "A high-performance, portable implementation of the MPI message passing interface standard", Parallel Computing, Vol. 22, pp. 789-828, 1996.
[2] Walker, D.W., "The design of a standard message passing interface for distributed memory concurrent computers", Parallel Computing, Vol. 20, pp. 657-673, 1994.
[3] Karrels, E. and Lusk, E., "Performance Analysis of MPI Programs", International Workshop "Environments and Tools for Parallel Scientific Computing", pp. 195-200, SIAM, 1994.
[4] Amza, C and Cox, A., "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, pp 18-28, Feb. 1996.
[5] Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele Jr. G.L., Zosel, M.E., *The High Performance Fortran Handbook*, The MIT Press, Cambridge, MA, (1994).
[6] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, (1994).
[7] Snir, M., Otto, S.W., Huss-Lederman S., Walker, D.W., and Dongarra, J, *MPI: The Complete Reference*, The MIT Press, Cambridge, MA, (1996).
[8] Pacheco, S.P., *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, CA, (1997).
[9] Foster, I.T., *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA, (1994).
[10]Bernecky, W. R., "Range-Focused k-ω Beamforming of a Line Array", NUWC-NPT Technical Memorandum 980118, Sept. 1998.
[11] http://www.mpi-forum.org, http://www.mcs.anl.gov/mpi, http://www.erc.msstate.edu/mpi